NPS52-82-006

NAVAL POSTGRADUATE SCHOOL

Monterey, California

A Relational Program for a Syntax Directed Editor

Bruce J. MacLennan

April 1982

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, Va 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Acting Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-82-006	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* A Relational Program for a Syntax Directed Editor		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER*(s)*
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; RR000-01-10 N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE April 1982 13. NUMBER OF PAGES 37
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/ DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Relational Programming, Functional Programming, Applicative Languages, Very-High-Level Languages, Syntax Directed Editor, Language-Oriented Editor, Structure Editor.

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report provides a basis for evaluating relational programming by presenting an implementation of a moderately complex program using relational programming. The program is a syntax-directed editor, a major component in a programming environment that allows the direct construction, modification, and display (unparsing) of parse trees. Relational programming is ideal for this application, since the relational calculus provides a number of high-level operators for manipulating trees and other complex data structures. The editor is presented in two notations: the usual mathematical notation and a sim-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

ple natural-language-like notation.

A RELATIONAL PROGRAM FOR A SYNTAX DIRECTED EDITOR*

B. J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93940

1. Introduction

An essential part of the investigation of any new programming method or programming language is the evaluation of its use in typical applications. In this report we present a use of relational programming [4, 3, 5] to implement a moderate size application, a syntax-directed editor.

The syntax-directed editor described in this report is similar in design to that described in [2]. This is a structure editor, that is, an editor which directly constructs and modifies a Parse tree. The source form of the program is unparsed from this tree onto the display whenever it is necessary to update the screen. Other than on the user's screen, the source form of the program is never explicitly stored, neither in memory nor on disk.

In addition to this unparsing function, the editor implements a number of language-independent commands for shifting the focus of attention among the nodes of the parse tree and for

* The work reported herein was supported by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

deleting and moving subtrees of the parse tree.

Both the program-entry and unparsing functions are syntax-directed; i.e., they are implemented as language-independent functions operating on an augmented BNF grammar for the language.

Each alternative in a rule of this grammar has an associated command key; by typing this key the user causes that alternative to be used to build a new part of the parse tree, provided, of course, that that alternative is syntactically legal at that point in the tree.

Since an editor is an interactive program that alters a data-base (the parse tree) in time, it is not appropriate to implement it in a purely applicative, or value-oriented, way (see [6] for a discussion of this). Hence, we have a few variable-like objects that can be updated by an assignment operation. The most important of these are T, the parse tree, and N, the current node. Object-oriented operations did not appear in previous descriptions of relational programming; they are still under investigation.

In the following sections we describe the data structures and relational definitions required to implement the syntax-directed editor. The relational notation is summarized in Appendix 1, although the reader unfamiliar with the concepts should consult [4] for a better introduction. The definitions constituting the syntax-directed editor are collected in Appendix 2. Since some potential users of relational programming may be

intimidated by its symbolic notation, we have developed an alternate natural-language-like notation, which is described in Appendix 3. Appendix 4 contains the syntax-directed editor translated into this more natural notation. The reader may be interested in comparing Appendices 2 and 4.

2. Description of Data Structures

In this section we will describe the data structures used by the syntax-directed editor and their representation as relations. The central data structure is, of course, the program tree, T. It is described below in terms of three subrelations,

$$T = Tu \mid Td \mid Tc.$$

Trees are composed of nodes and other values associated with the nodes. The Tc relation defines the interconnections between nodes. Thus, m = Tc(n, i) means that the i-th descendent of the node n is the node m. In other words, Tc is a mapping from node-integer pairs into nodes:

$$Tc \in (nodes \times ints) \rightarrow nodes$$

Each node in the tree has an associated value depending on whether the node is undefined or defined. An undefined node represents a part of the program that has not been entered, so it has a non-terminal name associated with it. This association is defined by the function Tu:

$$Tu \in nodes \rightarrow non\text{-}terms$$

Similarly, the function Td defines the mapping for defined nodes. For a defined node a particular alternative of a non-terminal has been selected; this alternative is identified by the key code that selects it. Therefore the Td function maps nodes into non-terminal name - key pairs:

$$Td \in nodes \rightarrow non\text{-}terms \times keys$$

For convenience, all three relations, Tc, Tu, and Td, are combined into one relation T which represents the program.

$$T = Tc \mid Tu \mid Td$$

This can be done because they all have disjoint domains. Notice that it is easy to recover Tc, Tu, and Td from T:

$$Tc = T \leftarrow nodes$$
$$Tu = T \leftarrow non\text{-}terms$$
$$Td = T \leftarrow non\text{-}terms \times keys$$

Next we will consider the representation of the grammar. Each non-terminal has associated with it a set of alternatives, each selected by a key character. If nt is a non-terminal name and k is a key character, then Alts(nt,k) is the rule for processing option k of non-terminal nt. Thus the type of Alts is:

$$Alts \in (non\text{-}terms \times keys) \rightarrow rules$$

Each non-terminal must also designate another, forwarding, non-terminal. If a non-terminal has no alternative corresponding to a key character, then the key character is forwarded to its

forwarding non-terminal. Forw(nt) is the forwarding non-terminal of nt; the type of Forw is:

$$Forw \in non\text{-}terms \rightarrow non\text{-}terms$$

Next we must address the representation of the grammar rules themselves. Each rule has three parts, an analysis part, a synthesis part, and a dictionary:

$$rules = analyses \times syntheses \times dictionaries$$

The analysis parts are just sequences of terminals and non-terminals:

$$analyses = (terms \mid non\text{-}terms)^2$$

The synthesis parts are trees that will be substituted into the appropriate part of the program tree T. Thus they have the same type as T:

$$syntheses = (nodes \mid nodes \times ints) \rightarrow (nodes \mid non\text{-}terms \mid non\text{-}terms \times keys)$$

The dictionary part of a rule is used to determine the component of a node associated with a particular non-terminal. It is generated by the grammar preprocessor when the Alts and Forw relations are generated.

$$dictionaries = non\text{-}terms \rightarrow ints$$

This completes the specification of the types of the program tree and the grammar. They are defined in terms of the primitive

type	definition
ints	integers
keys	key characters
nodes	nodes in program tree (atomic)
non-terms	non-terminal names
terms	terminal strings

Figure 1. Primitive Types

types listed in the following figure.

3. Editor Functions

3.1 Top Level of Refinement

Each entry of a key-stroke k must define a new state. Therefore, we will define a function 'process' such that process (k) will take the old state into the new:

$$s' = process\ k\ s$$

The type of 'process is:

$$process \in keys \rightarrow (states \rightarrow states)$$

We can define 'process' as the union of two functions: language-independent editing commands and language-dependent program-entry commands:

$$process = lang\_ind \mid lang\_dep$$

The language-independent processing function is just the union of pairs, each pair composed of an editing character and the function to perform the editing operation:

```
lang_ind  =  ( '↑':prev  |  '↓': next
             |  '+':succ  |  '-':pred
             |  '→':in    |  '←':out
             |  'G':get   |  'P':put
             |  'D':del   |  'U':undel )
```

The individual functions will be described below.

The language-dependent processing is performed by a function 'enter' which depends on the grammar. Since lang_dep(K) = enter (K), lang_dep = enter. This function is described later.

3.2 Positioning Commands

In this section we will describe the language-independent positioning commands which shift the focus of attention of the editor. The focus is represented by a variable N of type node. Each positioning command determines a new value of N based on the old value. To accomplish shifting the focus, we define move(f) which applies the positioning function f to the current node:

$$move(f) = N := f(N)$$

Consider first the 'out' command; this shifts the focus from a node to its parent. To accomplish this we need a function 'parent' defined so that parent(n) is the parent of node n. Suppose that n is the i-th descendent of m:

$$n = T(m,i)$$

We can invert this to:

$$(m,i) = T^{-1}(n)$$

Now the parent of n is just m, so

$$parent(n) = first.T^{-1}(n)$$

It is always possible that the user will try to move to the parent of the root, which doesn't exist. Therefore, if parent(N) is undefined we want 'out' to be an identity function. To accomplish this we define total(f) which makes any function f total by extending it with the identity function, total(f) = f/Id. The resulting definition of 'out' is:

```
total  =  (/Id)
parent =  first.T-1
out    =  move.total parent
```

The 'in' command moves to the first descendent of the current node. That is,

$$newN = T(oldN,1) = T.(,1)\ oldN$$

As for 'out', we want 'in' to be an identity if there is no first descendent, i.e.. if we are at a leaf. This results in the definition:

$$in = move.total\ T.(,1)$$

The 'next' and 'prev' commands move to the right and left siblings, respectively, of the current node. Thus we must say what it means for n to be the right sibling of m: n =

rightsib(m). This means that m and n have a common parent p such that m=T(p,k) and n=T(p,k+1). Thus (p,k) = $T^{-1}(m)$, so

$$n = T.(Id \,||\, (+1)).T^{-1}(m)$$

Now, the isomorphism of a relation R under a function f is defined:

$$f\$R = f^{-1}.R.f$$

so we can define the right sibling:

$$rightsib = T^{-1}\$(Id||(+1))$$

As we have said, the effect of 'next' is to move to the right sibling of the current node, and we have defined 'rightsib' to accomplish this. What if the current node doesn't have a right sibling? We could, as in the 'in' and 'out' commands leave the focus where it was. A better approach is to move the focus to the parent of the current node, and seek again for a right sibling. This process should continue until a node with a right-sibling is found, or we have reached the root of the tree. This is illustrated in the Figure 2.

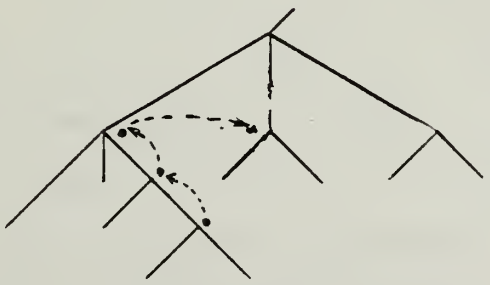


Figure 2. Effect of 'next' Command

The desired effect can be described as follows: as long as the current node has no right sibling, move to the parent, otherwise select the right sibling. This is easily expressed:

```
next = move.total [while( non.dom rightsib, parent); rightsib]
```

As usual, we have extended the function with 'total' to handle nodes for which 'next' would be undefined. The 'prev' operation is identical, except that '$rightsib^{-1}$' replaces 'rightsib'.

The remaining two positioning commands are 'succ' and 'pred'. These are used for moving to the succeeding and preceding members of a sequence. Their effect is shown in Figure 3. It can be seen that 'succ' is a 'next' followed by an 'in', and 'pred' is an 'out' followed by a 'prev':

```
succ = next; in
pred = out; prev
```

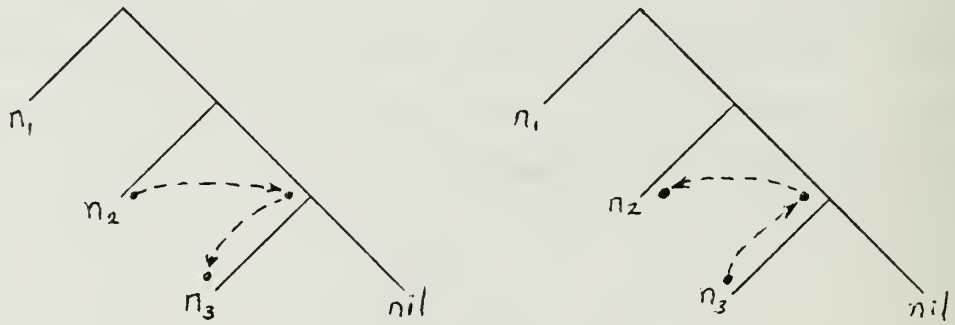


Figure 3. Effect of 'succ' and 'pred'

3.3 Editing Commands

There are really only two editing commands: deleting the subtree rooted at the current node, and inserting a new subtree at the current node. Each of these commands exists in two forms, as is

described below.

The get command deletes the subtree rooted at the current node, and saves it in the get-buffer. The put command reverses this operation by replacing the current node with the tree in the get-buffer. The delete command cperates the same as get, except the deleted subtree is placed in the save-buffer. This allows a later undelete command to reverse the effect of the delete. Undelete is just like put except that it uses the save-buffer.

To accomplish these functions we will define 'remove' and 'replace' which take as an argument the buffer.

```
get   =  remove G
put   =  replace G
del   =  remove S
undel  =  replace S
```

There are two steps in removing a subtree: (1) the subtree rooted at N, the current node, must be placed in the appropriate buffer. (2) this subtree must be deleted from the program tree:

```
remove(L)  =  L := subtree N; delete
```

Next we define 'subtree' and 'delete'.

The subtree rooted at a node n is just that portion of the program tree containing nodes reachable from n. Thus, if 'sub-nodes n' is the set of all nodes reachable from n, then

```
subtree(n)  =  (m | m X ints) → T
    where  m = subnodes n
```

To find the subnodes of n we will use a function 'reach' defined so that reach(S) is the set containing every node whose parent is in S:

$$reach(S) = img\ T\ (S\ X\ ints)$$

Hence,

$$reach = (img\ T).(X\ ints)$$

Then, to find the subnodes reachable from n we apply 'reach' zero or more times to the unit set containing n. Thus,

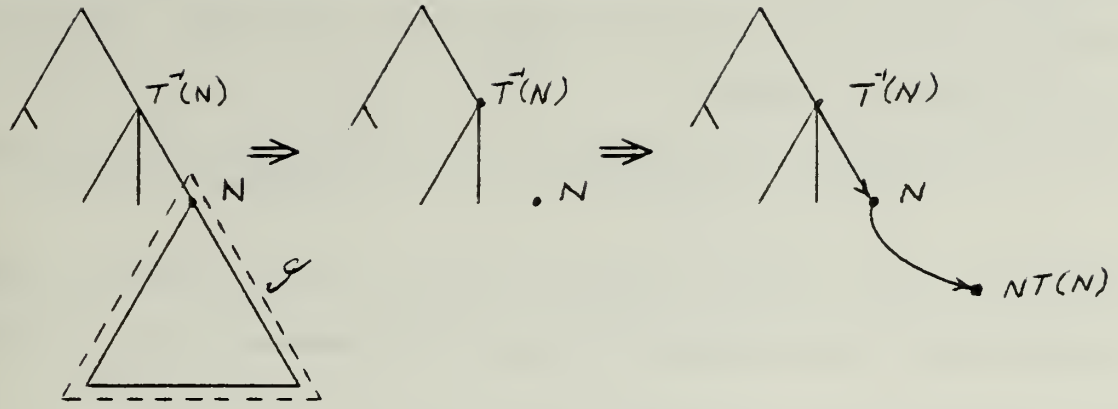
$$subnodes(n) = reach^{*}(un(n))$$

Therefore,

$$subnodes = reach^{*}.un$$

This completes the definition of 'subtree'.

The 'delete' function must remove all the nodes in subnodes(N). However, it also must replace the deleted subtree with the non-terminal expected at that point in the tree. It does this by creating an edge from the parent of N to N, and from N to the non-terminal associated with N. These operations can be visualized:



They are accomplished by:

delete = T := T <> non.subnodes N ¦ (T^{-1}N, N, NT N)

NT = first.T

The definition of NT comes about as follows: T(N) = (nt,k), the non-terminal key pair that generated node N. Hence, NT(N) = first.T(N). This completes the definition of 'remove'.

Replacing the current node (assumed to be undefined) with the contents of buffer L is quite simple: create a link from the parent of N to the root of L, and add L to the program tree:

replace(L) = T := (T^{-1}N : first L ¦ L) / T

The root of L is just its first number.

3.4 Program Entry Functions

In this section we will investigate the definition of the function 'enter(k)' which processes the language-dependent command key, k. If N is an undefined, or open, node, then T(N) is the

non-terminal associated with N, say, nt. This non-terminal is passed along with the command key k to a function 'select(nt,k)' for processing. Hence,

enter(k) = select (T N, n) = select.(T N,) n

Of course, the 'enter' function can only be applied to undefined nodes, so we will restrict select to undefined nodes:

enter = udf → select.(T N,)
where udf = img T^{-1} non-terms

Next we will consider the definition of 'select(nt,k)'. Recall that Alts.(nt,) is the mapping of command keys to alternatives for non-terminal nt. If k is in the domain of this mapping, then the associated rule is selected and processed:

if (nt,k) ∈ dom Alts
then process.Alts(nt,k)

If k is not handled by rule nt, then we must process the forwarding rule for nt, Forw(nt), and retry entering k. The resulting definition of 'select' is:

select(nt,k) =
  if (nt,k) ∈ dom Alts
  then process.Alts(nt,k)
  else process.Forw nt; enter k

= if (nt,k) ∈ dom Alts
  then process.Alts(nt,k)

```
else (;) [ process.Forw nt, enter k]
```

Notice that

```
[process.Forw nt, enter k]  =  (process.Forw || enter) (nt,k)
```

Hence,

```
select  =  [ process.Alts / (;).(process.Forw || enter) ]
```

The only function left to define is 'process', which handles the processing of a rule, i.e., which installs the synthesis part of a rule, which is the rule's second component. Hence, the tree to be inserted is t=new.second r, where r is the rule and 'new' creates a new copy of the tree. The function replace(t) will insert this new subtree. Finally, the cursor must be positioned at the first descendent in the new tree. Putting this all together:

```
process(r)  =  replace.new.second r; in
```

Hence,

```
process  =  in.replace.new.second
```

3.5 Unparsing

The last major function we must discuss is unparsing, i.e., the generation of source form of the program from the program tree. We will define a function unparse(n) which unparses the subtree rooted at node n. There are two cases: either node n is undefined or it is defined. If it is undefined then T(n) is the

non-terminal name associated with n, and this is what must be displayed. Otherwise we will use a function dispnode(n) to display a defined node:

unparse = udf →T / dispnode

The function of dispnode(n) is to display a defined node n; for this it is necessary to find the grammar rule that generated this node. Since n is defined, T(n) = (nt, k) where (nt, k) is the non-terminal name - key pair. If n was generated by an alternative, then 'Alts(nt,k)' is the rule. Otherwise, it was generated by a forwarding rule and 'Forw(nt)' is the rule.

The node n is unparsed according to rule r by disprule(n, r), defined later. We can now derive the definition of 'dispnode':

```
dispnode(n) = disprule(n,
  if (nt,k) ∈ Alts then Alts(nt,k)
  else Forw(nt) endif)
    where (nt,k) = T(n)

  = disprule(n,[Alts / Forw.first](nt,k))

  = disprule(n, [Alts / Forw.first].T (n))
```

Therefore,

dispnode = disprule.(Id # [Alts/Forw.first].T)

Disprule(n, r) takes a node n and a rule r and converts it to a

character string. It will do this using an auxiliary function Danal(n, r) which returns a sequence of strings, one corresponding to each item in the analysis part of r. These strings must be catenated to form the output of 'disprule'. Hence,

disprule(nt, r) = [cat @ ''](danal(nt, r))

Hence,

disprule = [cat @ ''].danal

Let's consider danal(n, r). The analysis part of rule r, first(r), is a sequence of items,

$$\langle a_1, a_2, \ldots, a_n \rangle$$

We wish to return an isomorphic sequence of strings,

$$\langle s_1, s_2, \ldots, s_n \rangle$$

such that each s_i is the result of displaying item a_i according to the current node. For the latter purpose we use a function disp(n, r, a_i). Thus,

$$s_i = \text{disp}(n,r,a_i)$$

Hence, the sequence s is just the image of a (the analysis part of r) under disp.(n, r,):

s = disp.(n,r,) $ a

So the definition of 'danal' is:

danal(n,r) = disp.(n,r,) $ first(r)

This brings us to 'disp'; disp(n,r,ai) displays item ai appropriately, i.e., if ai is a terminal it is displayed directly; if it is a non-terminal then the corresponding subnode of n is unparsed. The latter function is performed by dispnt(n,r,ai). Hence,

disp(n,r,ai) = dispnt(n,r,ai), if defined
ai, otherwise

The definition is

disp = dispnt / third

Finally, dispnt(n,r,ai) unparses the descendent of n corresponding to ai. Thus dispnt must perform unparse(T(n, k)), where k is the index of the descendent corresponding to ai. The index k is given by the "dictionary", or third, part of a rule, hence k = (third r) ai. This leads to the definition of 'dispnt':

dispnt(n,r,ai)
= unparse(T(n, third r ai))
= unparse.T(n, third r ai))

This completes the definition of the syntax-directed editor. All of the definitions of the functions are gathered in Appendix 2 and in the natural notation in Appendix 4.

4. References

[1] Brown, J.C., Loglan 4 & 5, A Loglan-English/English-Loglan Dictionary, 2nd. Ed., Palm Springs: 1975, The Loglan Institute, viii-xviii.

[2] MacLennan, B.J., The Automatic Generation of Syntax Directed Editors, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-014, October 1981.

[3] MacLennan, B.J., Introduction to Relational Programming, Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture, 213-220, October 18-22, 1981; also Naval Postgraduate School Computer Science Department Technical Report NPS52-81-008, June 1981.

[4] MacLennan, B.J., Overview of Relational Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-017, November 1981.

[5] MacLennan, B.J., Programming with a Relational Calculus, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-013, September 1981.

[6] MacLennan, B.J., Values and Objects in Programming Languages, Naval Postgraduate School Computer Science Department Technical Report NPS52-81-006, April 1981.

1. APPENDIX: SUMMARY OF RELATIONAL OPERATORS

Symbol	Meaning
x¦y	union of sets or relations
x&y	intersection of sets or relations
x~y	difference of sets or relations
x:y	ordered pair
x=y	equality
x:=y	assignment to variable
x/y	extension, = x ¦ (non.dom x → y)
(py)	bind right argument of p to y
(xp)	bind left argument of p to x
(p)	operator used as an operand
f^{-1}	inverse (converse)
non x	complement of set or relation
f.g	composition of f and g
f;g	relative product (reverse composition)
f x	functional application
x,y	sequence construction
f$r	isomorphic image of a relation
while(x,y)	iterative application
x X y	cartesian product
x → f	restrict domain
f ← x	restrict codomain
f<>x	restrict both domains
img f	image function
f^{*}	reflexive transitive closure

f^+	transitive closure
un	unit-set constructor
f||g	parallel application
f#g	construction
Id	identity function (equivalent to =)
f@x	f-reduction with initial value x
first	first member of relation (also second etc.)
dom f	domain of function or relation
$x \equiv y$	define x to be y
$x \in y$	set membership

2. APPENDIX: THE SYNTAX DIRECTED EDITOR

2.1 Top Level

```
process  =  lang_ind ¦ lang_dep

lang_ind  =  ( '↑':prev ¦ '↓': next
             ¦ '+':succ ¦ '-':pred
             ¦ '→':in ¦ '←':out
             ¦ 'G':get  ¦ 'P':put
             ¦ 'D':del  ¦ 'U':undel )

lang_dep = enter
```

2.2 Positioning

```
move(f)  =  N := f(N)
total  =  (/Id)

parent  =  first.T⁻¹
out  =  move.total parent
in  =  move.total T.(,1)

f$R  =  f⁻¹.R.f

rightsib = T⁻¹$(Id¦¦(+1))

next  =  move.total [while( non.dom rightsib, parent); rightsib

prev  =  move.total [while( non.dom rightsib⁻¹, parent); right

succ  =  next;  in
pred  =  out; prev
```

2.3 Editing

```
get   =  remove G
put   =  replace G
del   =  remove S
undel  =  replace S

remove(L)  =  L := subtree N; delete
subtree(n)  =  (m | m X ints) → T
    where  m = subnodes n
reach  =  (img T).( X ints)

subnodes  =  reach*.un

delete  =  T := T <> non.subnodes N | (T⁻¹N, N, NT N)
NT  =  first.T

replace(L)  =  T := (T⁻¹N : first L | L) / T
```

2.4 Program Entry

```
enter  =  udf →select.(T N,)
udf = img T⁻¹ non-terms
select  =  [ process.Alts / (;).(process.Forw || enter) ]
process  =  in.replace.new.second
```

2.5 Unparsing

```
unparse  =  udf →T / dispnode
dispnode  =  disprule.(Id # [Alts/Forw.first].T)
disprule  =  [cat @ ''].danal
```

```
danal(n,r)   =  disp.(n,r,) $ first(r)

disp   =  dispnt / third

dispnt(n,r,ai) =  unparse.T(n, third r ai))
```

3. APPENDIX: NATURAL NOTATION FOR RELATIONAL PROGRAMMING

In this appendix we present a less mathematical syntax for relational programming. By combining non-symbolic operator names with a right-associative, infix syntax, and comma and colon rules that suppress many parentheses, a natural, readable notation results. Of course, some of the manipulative advantages of a mathematical notation are lost.

Briefly, the syntax is as follows: All identifiers are divided into three classes: niladic (x, y, z, in the following examples), monadic (f, g), and dyadic (p, q, r). Monadic applications, whether functions or predicates, are written "f x", "f g x", etc. These associate to the right, hence "f g x" means "f(g x)". Dyadic applications, whether functions or relations, are written with a right-associative, infix syntax. That is, "x p y q z" means "x p (y q z)". Monadic applications are more binding than dyadic applications; hence, "f x p g y" means "(f x) p (g y)".

Commas and colons can be used to eliminate many parentheses. A comma is equivalent to a right parenthesis; the corresponding left parenthesis is at the nearest preceding colon, or at the beginning of the expression, if there is no preceding colon. Hence, "x p y, q z" means "(x p y) q z" and "x p: y q z, r w" means "x p (y q z) r w", which by right-associativity means "x p ((y q z) r w)". These rules have been inspired by the Loglan syntax [1].

Since the parsing of expressions is determined by the classification of identifiers into niladic, monadic, and dyadic, it is not possible to directly use a monadic or dyadic identifier as the argument to another application. To do this it is necessary to convert the monadic or dyadic identifier into a niladic identifier by quoting it. For example, the inverse of the dyadic identifier plus must be written

```
inverse 'plus'
```

The formal grammar for this notation follows. In the following appendix the syntax-directed editor is expressed in the natural notation.

3.1 Formal Syntax

```
assertion        =   expression.
expression       =   exp-head [ exp-tail ]
exp-head         =   { niladic-exp ¦ factor, }
factor           =   niladic-exp [ dyadic-exp factor ]
exp-tail         =   { dyadic-exp term ¦ dyadic-exp: expressi
term             =   niladic-exp [ exp-tail ]
niladic-exp      =   monadic-primary* niladic-primary
dyadic-exp       =   monadic-primary* dyadic-primary
niladic-primary  =   { niladic-id ¦ "(" expression ")"
                     ¦ ' { monadic-id ¦ dyadic-id } ' }
monadic-primary  =   { monadic-id ¦ "[" expression "]" }
dyadic-primary   =   { dyadic-id ¦ "{" expression "}" }
```

3.2 Vocabulary

Math. Notation	Natural Notation
x¦y	x combine y
x:y	x maps-to y
x=y	x equals y
x:=y	x becomes y
x/y	x extend y. x else y
(py)	something p y
(xp)	x p something
(p)	'p'
f^{-1}	inverse f
f;g	f then g
f x	f of x, x apply f
x,y	x;y, x connect y
f$r	f map r
while(x,y)	y do-while x
x X y	x cross y
x → f	x filter f, f if-in x
img f	image f
f^{*}	closure f
un	unit-set
f<>x	f restrict x
f¦¦g	f parallel g
f#g	f construct g, f also g
Id	identity
f@x	f reduce x

first	first
x+y	x plus y
dom f	domain f
$x \equiv y$	x means y

4. APPENDIX: SYNTAX DIRECTED EDITOR IN NATURAL NOTATION

4.1 SDE-Specific Vocabulary

Math. Notation	Natural Notation
N	current-node
G	move-buffer
S	save-buffer
T	tree
ints	integers
non-terms	non-terminals
Alts	alternation-rules
Forw	forwarding-rules

4.2 Top Level

```
Process means language-independent combine language-dependent.
Language-independent
   means: "↑" maps-to move-previous,
  combine "↓" maps-to move-next,
  combine "+" maps-to move-successor,
  combine "-" maps-to move-predecessor,
  combine "→" maps-to move-in,
  combine "←" maps-to move-out,
  combine "G" maps-to get,
  combine "P" maps-to put,
  combine "D" maps-to delete,
  combine "U" maps-to undelete.
```

Language-independent means enter.

4.3 Positioning

Move position-function means

  current-node becomes position-function of current-node.

Total means something extend identity.

Parent means inverse tree then first.

Move-out means parent apply total then move.

Move-in means: something maps-to 1, then tree, apply total then move.

Function map structure means

  function then structure then inverse function.

Right-sibling means inverse tree map identity parallel something plus

Move-next means parent do-while non domain right-sibling,

  then right-sibling, apply total then move.

Move-previous means parent do-while non domain inverse right-sibling,

  then inverse right-sibling, apply total then move.

Move-successor means move-next then move-in.

Move-predecessor means move-out then move-previous.

4.4 Editing

Get means remove-into move-buffer.

Put means replace-from move-buffer.

Delete means remove-into save-buffer.

Undelete means replace-from save-buffer.

Remove-from buffer means:

  buffer becomes subtree of current-node, then excise.

```
Subtree a-node means:
  tree if-in the-subnodes combine the-subnodes cross integers,
    where the-subnodes means subnodes of a-node.
Reach means:  something cross integers, then image tree.
Subnodes means unit-set then closure reach.

Excise means tree becomes
  tree restrict non subnodes of current-node
  combine: current-node apply inverse tree,
  connect current-node sequence non-term of current-node.
Non-term means tree then 'first'.

Replace-from buffer means tree becomes:
  current-node apply inverse tree, maps-to first buffer,
  combine buffer, extend tree.
```

4.5 Program Entry

```
Enter means: current-node apply tree,
  maps-to something, then select, if-in undefined-nodes.
Undefined-nodes means non-terminals, apply image inverse tree.
Select means: alternation-rules then process,
  else: forwarding-rules then process, parallel enter,
  then 'then'.
Process means 'second' then new then replace-from then move-in.
```

4.6 Unparsing

```
Unparse means: tree if-in undefined-nodes, else display-node.
Display-node means: identity construct
```

```
  tree then alternation-rules else 'first' then forwarding-rules,
  then display-rule.
Display-rule means display-analysis then 'catenate' reduce "".
Display-analysis (N; R) means: N connect R sequence something,
  then display, map first R.
Display means display-non-term else 'third'.
Display-non-term (N; R; non-term-name) means:
  N connect non-term-name apply R then 'third',
  apply tree then unparse.
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93940	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93940	1
Chairman, Code 52Bz Department of Computer Science Naval Postgraduate School Monterey, CA 93940	40
Professor Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, CA 93940	12
Professor Harvey Abramson Department of Computer Science The University of British Columbia 2075 Wesbrook Mall Vancouver, B. C. Canada V6T 1W5	1
Dr. M. Sintzoff Philips Research Laboratory 2 av. Van Becelaere 1170 Brussels Belguim	1
Dr. Mehdi Jazayeri Synapse Computer Corporation 801 Buckeye Court Milpitas, CA 95035	1
Mr. Jim Bowery Viewdata Corporation of America 1444 Biscayne Boulevard Suite 305 Miami, Florida 33132	1
Dr. Charles D. Marshall IBM Research, Department K51 5600 Cottle Road San Jose, CA 95193	1